SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 69–83 (1997)

Research

An Empirical Study of Software Reuse in Reconstructive Maintenance

WEI LI[1]

[1]*Computer Science Department, The University of Alabama in Huntsville, Huntsville, AL 35899, U.S.A.*

SUMMARY

Software reuse has mainly been studied in software development. However, software reuse is also an important aspect in software maintenance. Reconstructive maintenance involves disassembling an old system into components and reassembling them into a new one. This paper discusses software reuse in reconstructive maintenance. An empirical study in an industrial software development environment is presented, involving a legacy system. In particular, the study has found that the cyclomatic complexity metric at some levels may be related to a function's reusability. The study also found that there seems to be a high correlation between the cyclomatic complexity metric of a function and the lines of code metric of the function. **J. Software Maintenance 9: 69–83, 1997.**

(No. of Figures: 2. No. of Tables: 5. No. of Refs: 12.)

KEY WORDS: reconstructive maintenance; software reuse; software metrics; cyclomatic complexity, lines of code; empirical study

1. INTRODUCTION

Software has two major phases in its life span: development and maintenance. The border between the two phases is the first release of a software system. Software development creates a software product from conceptual ideas, which is also known as the forward engineering process. A software development process usually goes through requirements analysis, design and coding phases before the software is tested and released. The three phases in software development produce three models of the same system: requirements model, design model and implementation model. After the initial release of a software system, the subsequent changes to the three models of the system are dealt with as software maintenance.

Software maintenance is the life of software after the initial release. Clapp (1981) describes software maintenance as '. . .the phase in the life cycle of a system that follows software development. It is the period of time from the delivery of the system to its first user until the system is no longer used. . . .' Conger (1994) describes software maintenance as '. . .the period in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary. . . .'

CCC 1040–550X/97/020069–15$17.50

Received 21 October 1996
Revised 20 December 1996

The borderline between development and maintenance is not always clear. It is possible that a software system is constructed not entirely from scratch, but from an existing system. For example, a set of new software requirements are often initiated directly from existing ones. The construction of a new software system from existing ones is becoming frequent as more and more software products are being produced and maintained in industry. This kind of software production can neither fit into a traditional software development framework, nor be classified by the known maintenance categories. Such reconstructive software maintenance or development has the following characteristics:

- unlike a pure development, the new system has somewhat known requirements, design and implementation from the existing systems;
- the new system requirements include modified requirements from existing systems and new additions; and
- a large part of the new system can be assembled from existing components.

Software reuse is the use of existing software artefacts in the construction of new software systems. The reusable artefacts include source code, designs, requirements and other software product related components. Krueger (1992) used four dimensions to classify software reuse: abstraction, selection, specialization and integration. Abstraction refers to the abstract level of software artefacts. For example, design patterns (Gamma *et al*., 1995) are certainly more abstract than functions. Selection is the process of selecting software artefacts for reuse. Specialization makes a selected software artefact fit in the new system. For example, a new parameter may need to be added to a reused function in order to fit in the new system. Integration is the process of putting all the selected software artefacts together so that they work cohesively with newly constructed software artefacts in the new system.

Software reuse has mainly been studied in the context of object-orientated paradigm and forward engineering process (Krueger, 1992; McGregor and Sykes, 1992). Although the object-orientated paradigm provides an environment for developing more reusable software components than does the procedural paradigm, the reuse of existing system components which are not object orientated is also important economically and technically. The software systems designed and developed some time ago, also known as legacy systems, consume significant amounts of maintenance effort in industry. This paper discusses the reuse issues involved in the construction of a new system from a legacy system.

2. RECONSTRUCTIVE SOFTWARE MAINTENANCE

Different events can trigger a software maintenance request. Swanson (1976) classifies software maintenance activities into three categories: corrective, perfective and adaptive. Corrective maintenance is performed in response to software failure. Maintenance due to the change in data and processing environment is categorized as adaptive maintenance. Maintenance performed to eliminate processing inefficiencies, enhance performance or improve maintainability may be termed perfective maintenance.

Reconstructive maintenance is defined as the maintenance caused by some significant

changes in software requirements. The significant changes in software requirements may or may not be accompanied by a change in the software's operational environment. The objective of reconstructive maintenance is not to preserve an existing system, but rather to construct a new system from an existing system. It happens frequently when new products are designed and manufactured based on old products. Certain requirements and design features of the old products are preserved, and new requirements may be added. The changes to old software can be very drastic in software requirements, its operational environment, or both. The distinct characteristics of such software construction are: (1) the maintenance objective is to produce a new system from one or more existing systems; and (2) the new software has somewhat overlapped but different requirements from those of the existing software.

The reconstructive maintenance objective separates itself from the traditional categories of maintenance activities. The three traditional categories of software maintenance are aimed at preserving an existing system, while the goal of reconstructive maintenance is to build a new system. Some reconstructive maintenance may involve reverse engineering old software so that the old software's requirements and design can be better understood. However, reconstructive maintenance is independent of reverse engineering because it is not based upon and does not necessarily require reverse engineering.

The requirements of the new system in reconstructive maintenance are often tightly coupled with the old ones and may fall into one of three categories: (1) a scaled-up version of the old system; (2) a scaled-down version of the old system; or (3) a significantly modified old system. A scaling up from the old system requires adding significant amounts of new requirements and new design. A scaling down from the old system drops some old requirements, but new requirements may be added and design may still be needed. A drastically modified system may preserve relatively little of the old system. In general, deletions and additions of requirements for features and facilities are common in the modification process.

In describing the problems that a software sleuth commonly encounters, Kaliski and Kaliski (1991) listed, among other things, poor organization and division of software modules, and subtle and complex relationships between modules. They also suggested several sleuthing rules which included: (1) understanding the application; and (2) identifying the modular structure of a program, especialy the calling relations among modules and the relationships among functions and files. Although understanding the application domain and identifying modules are important steps in software maintenance, they are extremely important in reconstructive maintenance.

During reconstructive maintenance, the programmer/analyst must first understand the application domain as well as the old and new software requirements. Then, the software artefacts from the old system must be disassembled for possible reuse in the new system, and new software artefacts constructed. The integration process in reconstructive maintenance puts all artefacts, reused or newly constructed, together to work in the new system. These three major tasks motivated an iterative framework with incremental cycles to handle reconstructive maintenance.

3. A FRAMEWORK FOR RECONSTRUCTIVE MAINTENANCE

3.1. Three steps

Reconstructive maintenance is composed of three steps:

1. understanding the application domain,
2. sleuthing and disassembling existing software systems, and
3. constructing the new system.

The three steps are linked in an incremental cycle in the framework and are iterated many times depending on the new system size. The termination of the iteration in the framework is the completion of the new system. This framework is successfully employed in an industrial reconstructive maintenance project presented in the empirical study section of this paper. Figure 1 illustrates the framework of the reconstructive maintenance.

3.2. Understanding application domain

The process of learning the application domain knowledge in industry is common for programmers/analysts. There is a minimum set of knowledge that programmers/analysts must understand to work effectively in an application domain. For example, if a software system must be built to solve a problem in physics, accounting or servo-control, the programmer/analyst must possess a minimum amount of information about physics, accounting or servo-control in order to understand the requirements, and to design and implement the software. Walz, Elan and Curtis (1993) point out that 'software designers must be knowledgeable in the application domain'. If a programmer/analyst does not have the required knowledge, a learning process must be initiated. This learning process can best be achieved by combining top–down learning and bottom–up learning.

Top–down learning is the learning process employed by most educational institutions for training students in different disciplines. *Bottom–up learning* is learning through examples, observations, self study and peer discussions. Top–down learning is a formal learning process, while bottom–up is less formal, but with the emphasis on accomplishing

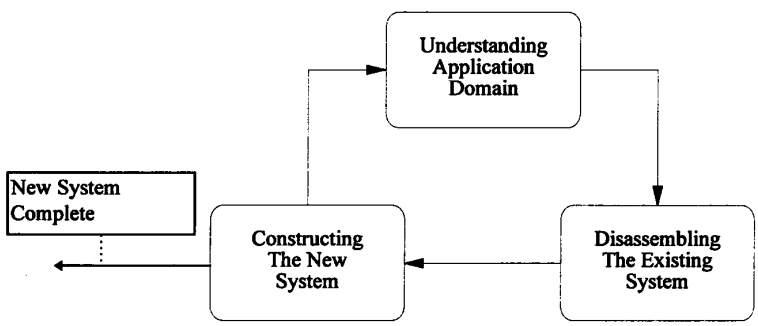

Figure 1. The incremental process cycle of the reconstructive maintenance

a specific task. For example, in order to write a program for a servo-controller, a programmer/analyst must gain the basic knowledge of control theory and different control techniques. If the knowledge is obtained by taking classes from a university or reading through several control theory books, the process of gaining the knowledge is top–down. If the knowledge is obtained by reading existing source code, documentation and peer discussion, then the process of gaining knowledge is bottom–up. Usually the best cost effectiveness in learning can be achieved through the combination of the two different types of learning processes. Walz, Elan and Curtis (1993) notice that top–down learning (formal sessions) is not very effective. 'These formal sessions did not seem to have much impact, primarily because the training did not focus on those things that were especially relevant for the project.'

3.3. Sleuthing and disassembling existing software systems

After the minimum set of application domain knowledge is obtained, a programmer/analyst must map the knowledge to functional modules in an existing software system. This process involves sleuthing an existing software system. During this process, a programmer/analyst first has to classify the concepts in the application domain into groups. The result is a set of groups of concepts. Each group is conceptually independent from other groups, but coherent within the group.

An identified group of concepts in the application domain is a *reconstructive concept set* (RCS). The contents of an RCS are normally based on the new system requirements. With RCSs identified, a programmer/analyst must then map each group onto a set of modules in the existing software. A collection of functional modules which implements an RCS is the *reconstructive module set* (RMS). There should be a one-to-one relationship from an RCS to an RMS. However, a particular functional module may be present in several RMSs, because the functional module may be used to implement several RCSs. The process of finding all the RMSs which correspond to the relevant RCSs is the second step in reconstructive maintenance. This process is also known as sleuthing and disassembling an existing system.

During the course of building the new system in reconstructive maintenance, the RCSs need to be identified. An RCS can be identified through the understanding of software requirements in the new system. After an RCS is identified, its corresponding RMS needs to be identified in the old system.

An RMS usually contains several functional modules, and a module contains several functions. When an RMS is identified as relevant to the RCS, which is needed in the construction of the new system, a programmer/analyst must decide which modules in the RMS can be reused in the new system. Typically, not all of the modules in an RMS are needed, especially when the new system is a scaled-down version of the old system. Of all the modules that are needed for the construction of the new system, only some will eventually be reused. For each module which is reused, modification to the module may be necessary. It is very rare that a reused module is not modified in the new system.

3.4. Construction of the new system

The construction of a new system is the process of incrementally implementing all the RCSs that are relevant to the new system. The incremental building of a new system exhibits a spiraling effect. First, a core RCS is picked to be implemented. The modules in the corresponding RMS are analysed for possible reuse. The reused modules are usually modified or customized for the new system. A set of new modules are often designed and implemented to work with the reused modules. The new system (with only the core RCS implemented) is tested to the point that it is reliable and robust enough to move to the next increment.

After the core RCS is implemented and tested, another RCS which is closely related to the core RCS is chosen to be designed and implemented next. This expansion of the new system indicates that the incremental building of the new system is spiraling into another cycle. The design and implementation of the second RCS goes through similar steps as in the core RCS process (analysing the corresponding RMS, choosing reusable modules, modifying reused modules and constructing new modules). Then, the expanded new system is tested and debugged. The inclusion of RCSs continues until all RCSs required by the new system are implemented and tested. It is important that only one RCS is included at each increment of the spiraling process.

It is worth noting that the incremental cycle illustrated in Figure 1 is based on RCSs. Each increment in the cycle would focus on one RCS. The cycle terminates when all the RCSs required in the new system have been included and implemented in the new system.

4. THE LAW OF DISASSEMBLE AND REASSEMBLE

In the context of a post-delivery maintenance phase, Clapp (1981) suggested three laws that govern the dynamics of program evolution. The ‘Law of Continuing Change’ describes the constantly changing nature of a software system. The ‘Law of Increasing Entropy’ describes the deterioration of a software system as more maintenance is performed. The ‘Law of Statistically Smooth Growth’ enables distinguishing between a change to software due to a specific external request, and a change due to the unbearable structure of the software because of repeated maintenance. While these laws are applicable to traditional software maintenance activities, they are not adequate to model the reconstructive maintenance. This paper offers the ‘Law of Disassemble and Reassemble’ to supplement the laws proffered by Clapp.

The ‘Law of Disassemble and Reassemble’ states that a new software system can be assembled based upon the components obtained by disassembling an existing software system into its constituent components. The requirements dictating the new system may overlap substantially the requirements descriptive of the old system. Such a new system is likely to be composed of modified existing components and newly constructed components. The single most important factor in the successful use of the existing components in new systems is the independence of the components. Provided that a module’s functionality suits the requirements of the new system, a loosely coupled cohesive module is easier to understand and more likely to be reused in a new system.

5. AN EMPIRICAL STUDY OF SOFTWARE REUSE

5.1. Metrics

Software metrics measure certain aspects of software. Software metrics can generally be divided into two categories: software product metrics and software process metrics (Li and Henry, 1993). Software product metrics measure software products, such as source code or design documents. Software process metrics measure the software development or maintenance process, such as the number of person-hours charged to the development activities in the design phase. This study explores the relationship between certain software product metrics and the reusability of a function in the reusable modules during the reconstructive maintenance.

A typical software size measure is the lines of code (LOC) metric. LOC is the count of source lines of code in a particular language implementation. For example, in the C programming language, the number of statements in a function is the LOC metric for the function. There are different ways to count a line of statement. This study uses the count of semicolons as the LOC metric.

The cyclomatic number was suggested by McCabe (1976) as a measure of the internal complexity of a function. The cyclomatic complexity (CC) metric counts the number of independent paths within a function. In fully-structured software, the CC is equal to one more than the number of single predicates in a function.

5.2. Software components

D1 is a servo position controller. D2 is a servo speed controller. D2 is a scaled-down version of D1 with additional requirements. D2's firmware is built through reconstructive maintenance from D1's firmware. D1's firmware is a mixture of C and assembly. D2's firmware is also composed of C and assembly. This study focuses only on the C source code of D1 and D2, because the assembly code is only a small portion of both D1's and D2's firmware. The reconstructive maintenance of D1's firmware leading to D2's firmware was about a two calendar-year project.

D1's firmware is divided into 11 modules. Each module is contained in a C source file. Table 1 summarizes D1's firmware. The 'module' column lists all the module names. The 'function count' column gives the number of functions within each module. The 'average LOC' and 'average CC' columns are the average of the LOC and CC metrics within each module. The 'correlation of CC and LOC' is the non-parametric Spearman's rank correlation coefficient (Ott, 1988) between the LOC and CC metrics for all the functions within a module.

D2's firmware was successfully reconstructed from D1's firmware by following the three-step framework described earlier. D2's firmware is a combination of reused artefacts from D1's firmware as well as new artefacts added to the system. Table 2 gives an overview of D2's firmware source code structure. The table structure of Table 2 is the same as for Table 1.

Table 1. D1's firmware source code structure

Module	Function count	Average LOC	Average CC	Correlation of CC and LOC
sconst.c	0	0·00	0·00	N/A
sd2.c	43	13·23	8·05	0·74
sdump.c	13	31·85	16·77	0·76
sedit2.c	46	11·09	5·22	0·76
serror2.c	5	25·4	12·40	0·81
seval.c	28	12·36	6·89	0·87
sformat.c	15	16·33	8·20	0·87
smotn2.c	29	17·34	8·48	0·90
ss.c	0	0·00	0·00	N/A
stune.c	1	144·00	49·00	N/A
sutil2.c	30	16·59	9·33	0·92

Table 2. Firmware source code metrics and statistics

Modules	Function counts	Average LOC	Average CC	Correlation of CC and LOC
ddcntl.c	42	22·52	8·20	0·75
ddconst.c	0	0·00	0·00	N/A
dderror.c	5	25·80	6·90	0·97
ddinit.c	20	15·00	5·53	0·67
ddpos.c	6	17·50	6·17	0·74
ddtune.c	1	34·00	46·00	N/A
ddutil.c	42	9·50	5·38	0·73

5.3. Structural complexity and function reusability

It is difficult to study reusability at the RMS level because the RMSs have a large overlap of modules within themselves. It is also difficult to study reusability at the module level because there is a large overlap of functions within the modules in different RMSs. Hence, for the most accurate and reliable result in this research on the reusability of a function, the function should serve as the basic measured unit that is reused during the reconstructive maintenance.

It is not difficult to speculate that a function with high internal structural complexity is less likely to be reused than a function that has low internal structural complexity. The high structural complexity of a function means that it is difficult to understand the behaviour of the function. Thus, it is reasonable to hypothesize that a function with a low CC value is more reusable than a function that has a high CC value. Hypothesis 1 is formed to test this belief.

Hypothesis 1

A function with a low CC value is more reusable than a function with a high CC value.

During the reconstructive maintenance of D1's firmware, a total of 45 functions were identified to be reusable in D2's firmware. However, only 31 out of 45 functions were actually reused in D2's firmware. Therefore, the entire group of reusable functions in D1 are divided into two samples, S1 and S2. S1 contains all the functions that were actually reused. S2 contains all the non-reused functions. The CC metric is calculated for each function in each sample.

In order to determine if the CC metric of a function influences its reusability, the sample means of S1 and S2 are compared in a Wilcoxon rank sum test (Ott, 1988). Speculatively, the average means in S1 should be less than in S2 because a programmer may avoid reusing a function that is too complex as indicated by the CC metric. If the CC metric has an impact on the reusability of a function, as speculated, the Wilcoxon test would indicate unequal sample means. Equal sample means, as a result of the test, would conclude that the CC metric does not have impact on the reusability of a function. The non-parametric Wilcoxon rank sum test is used because the data in S1 and S2 do not appear to be normally distributed. The null and the alternative hypotheses for the Wilcoxon rank sum test are given as follows:

H0. The two samples S1 and S2 have the same mean.

H1. The S1 sample has a lower mean than the S2 sample.

Table 3 summarizes the CC metrics in S1 and S2, as well as the ranks of each function in S1 and S2 according to the Wilcoxon rank sum test. Let T denote the sum of the ranks in S1. The test statistic z is calculated as: $z = (T - \mu_T)/\sigma_T$. The H0 rejection region is set at $\alpha = 0{\cdot}05$. The following summarizes the process of obtaining the test statistic z:

$$n1 = 31,\ n2 = 14,\ T = 670{\cdot}0000$$

$$\mu_T = [n1 \times (n1 + n2 + 1)]/2 = 713{\cdot}0000$$

$$\sigma_T^2 = (n1 \times n2) \times (n1 + n2 + 1)/12 = 1\ 663{\cdot}6667$$

$$\sigma_T = 40{\cdot}7881$$

$$z = -1{\cdot}0542$$

The critical z value at $\alpha = 0{\cdot}05$ is $-2{\cdot}5700$. Because the test statistic z value is larger than the critical z value, the Wilcoxon rank sum test failed to reject the null hypothesis.

Hypothesis H1 can be tested more directly by considering various levels of CC values at which it might be true. As shown in Table 3, seven, or 50%, of the S2 sample had CC values of six or less, while 22, or 71%, of the S1 sample had CC values of six or less. A binomial test yields a z value of $-1{\cdot}7284$ which is equivalent to a one-tailed probability of $0{\cdot}042$. With $\alpha = 0{\cdot}05$, this test accepts the H1 hypothesis for CC values of six or less.

Table 3. Summary of reused and not reused functions in D1

CC of S1	Rank of S1	CC of S2	Rank of S2
1	2·5	1	2·5
1	2·5	2	8
1	2·5	2	8
2	8	2	8
2	8	5	24·5
2	8	5	24·5
2	8	5	24·5
3	13	7	30
3	13	8	31
3	13	27	37
4	18	31	39
4	18	32	40
4	18	42	43
4	18	63	45
4	18		
4	18		
4	18		
5	24·5		
5	24·5		
5	24·5		
6	28·5		
6	28·5		
9	32·5		
9	32·5		
15	34		
21	35·5		
21	35·5		
28	38		
33	41		
41	42		
49	44		

These leave moot the original speculation that the more complex a function's internal structure is, the less likely the function is reusable. However, the independence of a function's reusability from its structural complexity makes sense. Because all the functions that are reusable are tested to be working properly in D1, there is very little need to decompose a reusable function in order to use it in D2. The reusability of a function in the circumstance is probably dominated by other factors rather than the structural complexity of the function. Possible factors that influence the decision to reuse or not to reuse a function are: (1) how well the function satisfies the requirements; and (2) how well the function fits the RCS's requirements in the current increment.

5.4. The relationship between LOC and CC

The LOC metric and CC metric are measuring different aspects of a function. The LOC metric is a function-size measure, while the CC metric is a logic-control complexity

Table 4. Correlation between LOC and CC in D1's firmware

Module	Sample size	Correlation	significance (p)
sd2.c	43	0·74	$p < 0{\cdot}05$
sdump.c	13	0·76	$p < 0{\cdot}05$
sedit2.c	46	0·76	$p < 0{\cdot}05$
serror2.c	5	0·81	N/A
seval.c	28	0·87	$p < 0{\cdot}05$
sformat.c	15	0·87	$p < 0{\cdot}05$
smotn2.c	29	0·90	$p < 0{\cdot}05$
sutil2.c	30	0·92	$p < 0{\cdot}05$

measure in a function. Common sense would presume that the two metrics should not correlate. This common sense assumption is tested in this project.

There are 11 C source modules in D1 and five C source modules in D2. The Spearman's rank correlation (Ott, 1988) for the average LOC and the average CC is calculated in each module of D1 and D2. A *t*-test is performed for each correlation to check the statistical significance of the correlation. The results are presented in Table 4 and Table 5. The tables show the correlation between LOC and CC in each module and the correlation's statistical significance using a *t*-test for D1 and D2, respectively.

There appears to be high correlation at 0.05 significance level between the LOC and CC metrics in the modules of D1 and D2, if we drop the statistical outliers whose sample size is less than ten. This somewhat surprising result indicates that the LOC and CC metrics, although measuring different aspects of a function, were statistically indistinguishable in this study. This result may be caused by the fact that, in practice, each function is constructed with reasonable logic-control structure so that a longer program means a more complicated internal logical control. However, this conclusion is very preliminary and is limited to one study of two systems only. Generalization of this result should be avoided. More studies of the relationship between LOC and CC in more diverse contexts are needed to further understand the relationship between the LOC and CC metrics.

Table 5. Correlation between LOC and CC in D2's firmware

Module	Sample size	Correlation	significance (p)
ddcntl.c	42	0·75	$p < 0{\cdot}05$
dderror.c.c	5	0·97	$p < 0{\cdot}5$
ddinit.c	20	0·67	$p < 0{\cdot}05$
ddpos.c	6	0·74	N/A
ddutil.c	42	0·73	$p < 0{\cdot}05$

5.5. Specialization of the reused functions

As noted earlier, Krueger (1992) discusses software reuse using the taxonomy of abstraction, selection, specialization and integration. The specialization of reusable software artefacts may take different forms. For example, an instantiation of a parametrized class is one form of specialization that does not involve any change to the original class. The modification of an artefact to fit into a new system is another form of specialization. The specialization of each reused function in reconstructive maintenance only takes the form of modification of each reused D1 function to fit into D2.

Figure 2 shows the specialization effort in the reconstructive maintenance. The 'total' figure is the total number of functions in D1. The 'reused' figure shows the number of functions which are reused in D2. The 'modified' number is the total number of reused functions which are modified. The 'not modified' figure indicates the total number of reused functions that are not changed.

As Figure 2 shows, only a small percentage of the entire function pool is reused (21%). But more than two thirds (69%) of the reused function are modified. This result may come from two factors: (1) the original system was not designed and implemented with careful consideration for reuse in the application domain; (2) the reuse abstraction level is too low. This result warns that the reuse of software artefacts at a code scavenging level always involves heavy specialization effort.

5.6. Other observations

In the course of understanding the application domain, both top–down and bottom–up learning approaches are employed to gain the understanding of the application domain. The bottom–up approach proves to be more effective than the top–down approach. This observation agrees with the findings by Walz, Elan and Curtis (1993) who summarized an industrial experience as: 'we were surprised to see how important context-sensitive learning was to the design process'. The understanding of the application domain leads to the identification of the RCSs needed in D2. The Appendix shows all the reconstructive concept sets (RCS) identified in the process.

There has not been any domain analysis which leads to the abstractions of some key common features in D1's firmware. Therefore, it was impossible to reuse software based on design patterns during reconstructive maintenance of a legacy system. There was no software reuse at the design level either, due to insufficient design documents. The

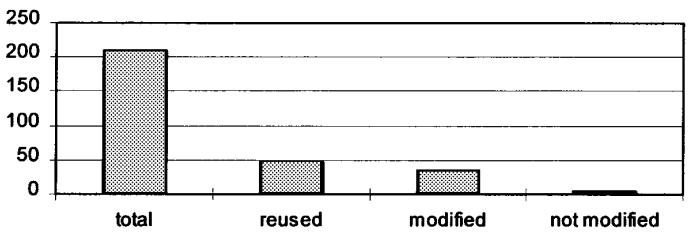

Figure 2. Reuse activity in D1's firmware

software reuse was mainly at the source code level. Certain functions are copied from the old system to the new system.

The abstraction of software artefacts (functions) must be created by the programmer/analyst. The reconstructive concept set (RCS) can help a programmer/analyst pick the abstractions out even without any design documents. The observed reuse activity is similar to the code scavenging in the findings of Krueger (1992). There does not seem to have been any similarity between the abstractions of D1's firmware and D2's firmware. This suggests that each designer approaches essentially the same application domain with different views. Abstraction depends on how a designer views the application domain, not on the application domain itself. It is very difficult to reuse a software artefact without modification if the level of abstraction is function based.

The selection of abstractions to reuse is driven primarily by the new system requirements, namely the RCSs. Since creating design patterns or higher level reusable abstractions are not the goals of this project, a function is selected from the old system for reuse in the new system mainly based on whether the function would satisfy, in part, a requirement demanded by an RCS.

Integrating a reused function into a new system requires changes in either the reused function of the new system context (Krueger, 1992). However, the new system context is created as reused functions are modified because the new system design goes in parallel with the code scavenging effort. Newly created artefacts tend to dominate in functionality as well as in numbers in the new system.

6. CONCLUSIONS

This paper has identified reconstructive maintenance as a way of constructing a new system from existing ones. The RCS and RMS concepts are proposed to guide reconstructive software maintenance. A three-step framework for performing reconstructive maintenance is also proposed.

An empirical study of reconstructive maintenance yielded the following results:

1. There is some evidence that lower levels of the internal structural complexity of a function, such as six or less, as measured in CC, may increase the reusability of a function in reconstructive maintenance.
2. There is a heavy specialization effort required for reuse in reconstructive maintenance.
3. There appears to be a high correlation at 0.05 significance level between the CC and LOC metrics of the module.

The observations in the empirical study also reveal the following:

1. There is a one-to-one relationship between RCS and RMS.
2. The RCS is an effective guide in the incremental construction of a new system during the reconstructive maintenance.
3. Software reuse in a legacy system during reconstructive maintenance is only at the code scavenging level. Design patterns and design reuse are nearly impossible during the reconstructive maintenance of a legacy system.

It is important to note that all of the above results and observations are very preliminary. Further research is needed to enhance the understanding of reconstructive maintenance. More empirical studies in various environments are needed to verify the results and observations obtained in this study.

APPENDIX. RECONSTRUCTIVE CONCEPT SETS IN EMPIRICAL STUDY

1. Angle advance (AA)
2. Automatic drive tuning (ADT)
3. Commutation (C)
4. Current foldback control (CFC)
5. Dual gain handling (DGH)
6. Electronic gear box (EGB)
7. Error handling (EH)
8. Filters (F)
9. Initialization (I)
10. Miscellaneous (M)
11. Position loop compensation (PLC)
12. Position loop acceleration and deceleration control (PLADC)
13. RAM test (RT)
14. ROM checksum test (RCT)
15. Serial I/O (SIO)
16. System main control (SMC)
17. System tuning (ST)
18. User command handling (UCH)
19. Velocity acceleration and deceleration control (VADC)
20. Velocity loop compensation (VLC)
21. Xilinx setup (XS)

Acknowledgement

I would like to thank the anonymous reviewers. Special thanks are due to Dr. Dwight A. Haworth for his comments in the earlier version of this paper.

References

Clapp, J. (1981) 'Designing software for maintainability', *Computer Design*, **20**(9), 197–202.

Conger, S. A. (1994) *The New Software Engineering*, Wadsworth Publishing Company, Belmont, CA, 684pp.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley, Reading, MA, 395pp.

Kaliski, M. E. and Kaliski, B. S. (1991) *The Software Sleuth*, West Publishing Company, New York, NY, 226pp.

Krueger, C. W. (1992) 'Software reuse', *Computing Surveys*, **24**(2), 131–183.
Li, W. and Henry, S. (1993) 'Object-oriented metrics which predict maintainability', *Journal of Systems and Software*, **23**(2), 111–122.
McCabe, T. J. (1976) 'A complexity measure', *Transactions on Software Engineering*, **SE-2**(4), 308–320.
McGregor, J. D. and Sykes, D. A. (1992) *Object-oriented Software Development: Engineering Software for Reuse*, International Thomson Computer Press, Boston, MA, 315pp.
Ott, L. (1988) *An Introduction to Statistical Methods and Data Analysis*, third edition, PWS-Kent Publishing Company, Boston, MA, 472pp.
Swanson, E. B. (1976) 'The dimensions of maintenance', in *Proceedings of the Second International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 492–497.
Walters, R. C. and Chikosfky, E. (1994) 'Reverse engineering progress along many dimensions', *Communications of the ACM*, **37**(5), 22–24.
Walz, D. G., Elan, J. J. and Curtis, B. (1993) 'Inside a software design team: knowledge acquisition, sharing, and integration', *Communications of the ACM*, **36**(10), 63–77.

Author's biography:

Wei Li is an Assistant Professor in the Computer Science Department at the University of Alabama in Huntsville. He has research interests in software metrics, object-orientated design, software reuse and software processes. His recent work has focused on measuring software reusability through quantitative metrics. He holds a Ph.D. in Computer Science from Virginia Polytechnic Institute and State University. Dr. Li is a member of the ACM and the IEEE. Email: wli@cs.uah.edu

